

SOFTWARE VERIFICATION RESEARCH CENTRE

SCHOOL OF INFORMATION TECHNOLOGY

THE UNIVERSITY OF QUEENSLAND

Queensland 4072

Australia

TECHNICAL REPORT

No. 98-29

**Supporting Contexts in the Sequential
Real-Time Refinement Calculus**

Luke Wildman and Ian Hayes

December 98

Phone: +61 7 3365 1003

Fax: +61 7 3365 1533

<http://svrc.it.uq.edu.au>

Proceedings of, Jim Grundy and Martin Schwenke and Trevor Vickers editors, International Refinement Workshop and Formal Methods Pacific 1998, 29 September - 2 October, Canberra, Australia, Discrete Mathematics and Theoretical Computer Science, Springer-Verlag, 981-4021-16-4, pgs 352-369

Note: Most SVRC technical reports are available via anonymous ftp, from [svrc.it.uq.edu.au](ftp://svrc.it.uq.edu.au) in the directory `/pub/techreports`. Abstracts and compressed postscript files are available via <http://svrc.it.uq.edu.au>

Supporting Contexts in the Sequential Real-Time Refinement Calculus

Luke Wildman and Ian Hayes

Abstract.

Program window inference provides support for contextual information such as preconditions and the scope and types of variables during derivations in the refinement calculus. Sequential real-time programs may be developed using a real-time refinement calculus in which variables are modelled by their traces over time. We introduce support for the contextual information required for derivations in the real-time refinement calculus, providing integrated support for the specification, refinement, and validation of sequential real-time programs.

1 Introduction

The refinement calculus [Bac80, Mor94] enables the systematic derivation of programs from specifications via a series of correctness preserving transformations. The real-time refinement calculus [Mah92] allows the specification of real-time systems and their subsequent refinement to parallel processes. In this approach, variables are modelled by their traces over time. Utting and Fidge [UF96] have extended this work to allow formal specifications with real-time requirements to be refined to sequential programs annotated with precise timing constraints. Based on that work, a sequential real-time refinement calculus [HU97] has been defined which provides refinement steps for the separation of timing constraints from the functional requirements and for which the refinement of the functional requirements is essentially the standard refinement calculus.

With an aim to develop tools supporting the refinement of real-time specifications to sequential programs annotated with timing constraints, we examine the application of Program Window Inference [NH95, NH97] as currently used in the Program Refinement Tool [CHN⁺96]. Program window inference provides support for contextual information during derivations in the refinement calculus.

2 Program Window Inference

Window Inference [RS93,Gru96] proposes a practical, goal-directed paradigm for interactive theorem proving. At each step the user is presented with the problem to be solved, F , the *focus*. The goal is to evaluate or simplify F , or more generally to transform F to a related V via preorder e (a reflexive and transitive relation). Also provided are hypotheses, H , which may be used in the transformation. Together F , e , and H form a *window*. A window defines an intention to prove a result of the form $H \Rightarrow (F e V)$ for an expression V . For example, to prove a theorem of the form $H \Rightarrow F$ we can use a window of the form $H \Rightarrow (F \leftarrow true)$ in which we transform F to $true$ under the relation reverse implication assuming hypothesis H . Similarly a window can be used to represent a refinement of S to S' with hypothesis H : $H \Rightarrow (S \sqsubseteq S')$. Window inference thus provides a single framework for both theorem proving and refinement, and in both, it provides support for contextual information via the hypothesis H .

A focus, $C[F]$, may be transformed by transforming one of its components, F . This is achieved using a window *opening* rule with the component to be transformed as its focus, F . The opening rule describes how to change the context (hypotheses) and the relation in the sub-window (premiss) in order to preserve the correctness of the parent window (conclusion). The opening rule below states that focus $C[F]$ can be transformed via relation e to $C[V]$ in context H , provided that we can transform focus F to V via relation e' in context H' .

$$\frac{H' \Rightarrow (F e' V)}{H \Rightarrow (C[F] e C[V])} \quad [\textit{open}]$$

A *focus transformation rule* is used to transform a focus directly. In general, to transform the focus F to V , preserving relation e , a transformation rule with conclusion $A e B$ must be found which unifies with $F e V$ and which has premises P which may be discharged by the hypotheses.

$$\frac{H \Rightarrow P}{H \Rightarrow (A e B)} \quad [\textit{transform}]$$

Program window inference [NH97] introduces support for other forms of contextual information corresponding more closely to the programming context: *l-value* information about variables where we refer to the names of the variables and not their values, i.e., *invariant* information about

For fresh names v' , u' , and x' ,

$$\begin{array}{c}
\frac{\Gamma; \text{pre } P \Vdash \boxed{S} \sqsubseteq S'}{\Gamma \Vdash \{P\}\boxed{S} \sqsubseteq \{P\}S'} \quad [\text{pre}] \\
\\
\frac{\Gamma; \text{inv } I \Vdash \boxed{S} \sqsubseteq S'}{\Gamma \Vdash \llbracket \text{inv } I \bullet \boxed{S} \rrbracket \sqsubseteq \llbracket \text{inv } I \bullet S' \rrbracket} \quad [\text{inv}] \\
\\
\frac{\begin{array}{l} \Gamma \\ \text{pre } P[v \setminus v'] \\ \text{inv } I[v \setminus v'] \wedge v \in T \quad \Vdash \boxed{S} \sqsubseteq S' \\ \text{lval } L[v \setminus v'] \wedge v \in \text{Var} \end{array}}{\Gamma; \text{pre } P; \text{inv } I; \text{lval } L \Vdash \llbracket \text{var } v : T \bullet \boxed{S} \rrbracket \sqsubseteq \llbracket \text{var } v : T \bullet S' \rrbracket} \quad [\text{var}] \\
\\
\frac{\begin{array}{l} \Gamma \\ \text{pre } P[u \setminus u'] \\ \text{inv } I[u \setminus u'] \wedge u \in T \quad \Vdash \boxed{S} \sqsubseteq S' \\ \text{lval } L[u \setminus u'] \wedge u \in \text{Con} \end{array}}{\Gamma; \text{pre } P; \text{inv } I; \text{lval } L \Vdash \llbracket \text{con } u : T \bullet \boxed{S} \rrbracket \sqsubseteq \llbracket \text{con } u : T \bullet S' \rrbracket} \quad [\text{con}] \\
\\
\frac{\Gamma; \text{pre } P[\tilde{x} \setminus \tilde{x}'] \Vdash \boxed{R} \Leftarrow R'}{\Gamma; \text{pre } P \Vdash \tilde{x} : \llbracket \boxed{R} \rrbracket \sqsubseteq \tilde{x} : [R']} \quad [\text{post}]
\end{array}$$

Fig. 1. Program window opening rules.

variable types and other local invariants, and *precondition* information about the values of variables in an initial state.

Some example program window opening rules are presented in Figure 1. The box indicates the focus. Rule *pre* is used for opening a window on a refinement which assumes precondition P , where Γ summarises the original context. In the premiss, the precondition information is moved into a hypothesis annotated *pre*. It represents contextual information about the pre-state of the command being refined. The *pre* rule effectively defines the *pre* context. Rule *inv* defines the *inv* context. The form of the premiss is analogous to the way in which Morgan and Vickers [MV90] write a sequence of refinement steps in the context of a local invariant I : $S \sqsubseteq_I S'$.

Rule *var* illustrates the use of l-value and invariant information. Opening a program window on the body of a local variable block causes the context to be updated with information about the declaration and type of the variable. Any occurrences of the variable name in the original hypotheses are renamed to a fresh name v' , and new hypotheses are added declaring the new name v as a variable (an l-value hypothesis annotated lval), and recording its type T (a local invariant annotated inv). In the lval hypothesis, v is interpreted as an identifier and Var is a set of identifiers. Whereas in both pre and inv hypotheses, v is interpreted as the value of the variable v in the appropriate state. A pre hypothesis holds in the pre-state of the command, and an inv hypothesis holds for all states.

The rule *con* for opening a program window on a local constant block illustrates an lval hypothesis declaring u as a logical constant. By differentiating constants and variables we can, for example, restrict the left-hand side of an assignment statement to only contain variables.

Rule *post* illustrates a program window opening rule which not only updates the context but also changes the relation between the focus and the goal. This rule encodes *strengthen postcondition* by renaming (to a fresh name \tilde{x}') occurrences of the frame variables in the preconditions (frame variables refer to the post-state in the post-condition) and replacing the refinement relation by reverse implication between postconditions.

A *refinement rule* describes a program window transformation with the associated proof obligations being the antecedants. The antecedants are annotated according to the type of hypotheses used to discharge them. For example, the following rule is used to introduce an assignment statement.

$$\frac{\Gamma; \text{pre } P; \text{inv } I; \text{lval } L \Vdash \begin{array}{l} \text{lval } \tilde{x} \subseteq \tilde{w} \\ \text{pre } \text{post}[\tilde{x} \setminus \tilde{e}] \end{array}}{\Gamma; \text{pre } P; \text{inv } I; \text{lval } L \Vdash \tilde{w} : [\text{post}] \sqsubseteq \tilde{x} := \tilde{e}} \quad [\text{ass}]$$

The lval proof obligation can be discharged using lval hypotheses only, as it requires only l-value information from the context. The pre proof obligation may require use of all the context information. For example, the above requires that the pre-state, state invariant, and l-value information imply that the assignment will produce the desired post-condition. An inv proof obligation must be discharged without use of pre information.

Note that the premisses and conclusion in the above example have the same form. In the rest of this paper we will abbreviate by leaving the

duplicated context implicit, e.g., the above rule is abbreviated as follows.

$$\frac{\text{lval } \tilde{x} \subseteq \tilde{w} \quad \text{pre } \text{post}[\tilde{x} \setminus \tilde{e}]}{\tilde{w} : [\text{post}] \sqsubseteq \tilde{x} := \tilde{e}} \quad [\text{ass}]$$

3 Supporting variables in real-time refinement

In the following sections we examine the novel features of the real-time refinement calculus for the purpose of providing contextual support in a real-time program window inference system. In the real-time refinement calculus, variables are treated as traces (functions of time), allowing a variable to be modelled by its behaviour over time [Mah92]. For example, the environmental variable *level* models the change in the level of water in a mine over time and the boolean variable *pump* is used to control a mine pump.

$$\begin{aligned} \text{level} &: \text{Time} \rightarrow \text{Depth} \\ \text{pump} &: \text{Time} \rightarrow \text{Boolean} \end{aligned}$$

Time and *Depth* are modelled by the non-negative real numbers.

Variables can be one of three kinds: input variables (controlled by the environment), and output and local variables (controlled by the program). They are similar in that they are all functions of *Time* however they are differentiated by the calculus. Input variables can not be assumed to remain stable just because the program does not modify them. Output variables refer to elements of the environment over which the program has control but local variables are visible in the program only and have no direct relationship with the environment. We distinguish the variables directly modifiable by the program (output and local but not input) by the term *program variables*. Program constants and logical constants may also be used. A program constant represents a defined value which can be implemented in a programming language. A logical constant represents some abstract value in the specification. For example, program constants *limit* and *minlevel* represent the upper and lower boundaries of the level of water in the mine, logical constant *u* represents a particular time occurring in the specification of the pump controller. Constants do not vary over time.

$$\begin{aligned} \text{const } \text{limit}, \text{minlevel} &: \text{Depth} \\ \text{con } u &: \text{Time} \end{aligned}$$

To support information about the types of variables and constants in the context of a real-time refinement, we introduce the following environment containing disjoint sets of names for each class of variable and constant.

$\begin{array}{l} \textit{Env} \\ \hline \textit{Output}, \textit{Input}, \textit{Var}, \textit{Const}, \textit{Con}, \textit{Names} : \mathbb{P} \textit{Ident} \\ \hline \textit{pairwise_disjoint}(\langle \textit{Output}, \textit{Input}, \textit{Var}, \textit{Const}, \textit{Con} \rangle) \\ \textit{Names} = \textit{Output} \cup \textit{Input} \cup \textit{Var} \cup \textit{Const} \cup \textit{Con} \end{array}$
--

We define new annotations corresponding to the declaration of real-time variables and constants.

$$\begin{aligned} \textit{output } v : T &\hat{=} \textit{lval } v \in \textit{Output}; \textit{inv } v \in \textit{Time} \rightarrow T \\ \textit{input } v : T &\hat{=} \textit{lval } v \in \textit{Input}; \textit{inv } v \in \textit{Time} \rightarrow T \\ \textit{var } v : T &\hat{=} \textit{lval } v \in \textit{Var}; \textit{inv } v \in \textit{Time} \rightarrow T \\ \textit{con } u : T &\hat{=} \textit{lval } u \in \textit{Con}; \textit{inv } u \in T \\ \textit{const } u : T &\hat{=} \textit{lval } u \in \textit{Const}; \textit{inv } u \in T \end{aligned}$$

Note that all variables are time dependent and so it is convenient to abbreviate the type to the type of the range of the variable. For example,

$$\textit{input level} : \textit{Depth} \hat{=} \textit{lval level} \in \textit{Input}; \textit{inv level} \in \textit{Time} \rightarrow \textit{Depth}.$$

4 Supporting assertions

Assertions in the real-time refinement calculus allow statements about variables over time. For instance, the pump decreases the water level whenever it has been turned on for at least one second. This property is expressed by the following assertion expressing an invariant over all time. In the real-time refinement calculus, commands are prefixed with a ‘ \star ’ to differentiate them from the corresponding standard refinement calculus commands.

$$\star\{\forall t : \textit{Time} \bullet \textit{pump}(\textit{[-}t-1 \dots t\textit{-]}) = \{\textit{true}\} \Rightarrow \frac{d}{dt}\textit{level}(t) < 0\}$$

The notation $\textit{[-}t-1 \dots t\textit{-]}$ stands for the closed interval of real numbers ranging from $t-1$ upto t , inclusive, and $\textit{pump}(S)$ is the set of all values of $\textit{pump}(x)$ for $x \in S$. We must also assume that \textit{level} is differentiable.

An assertion may also be used to state an *assumption* about the environment at the current time. The current time is represented by the

special variable τ of type *Time* which is itself *not* represented by a function of time. For example, the specification of a mine pump controller *assumes* that the water is initially above the limit.

$$\star\{level(\tau) > limit\}$$

A time dependent variable v is evaluated at time τ by *indexing*, i.e., $v(\tau)$. However, many references to time-dependent variables in the assumption refer to values of the variable at time τ . A convention is introduced so that unindexed references to a time-dependent variable v stands for $v(\tau)$ in the assumption. For example, the assertion above can be written: $\star\{level > limit\}$. This convention is formalised by the notation introduced by Hayes and Utting [HU97]. For a predicate P in the assumption, $P@x$ stands for P with all unindexed references to time-dependent variable v replaced by $v(\tau)$. This notation is generalised by the following definition.

Definition 1 (at time) For a predicate or expression P , $P@x$ is defined to be P with all unindexed occurrences of a variable v replaced by $v(x)$ and any occurrences of τ replaced by x .

A real-time assertion $\star\{P\}$ is semantically equivalent to an assertion in the standard refinement calculus of the form $\{P@x\}$. It can be supported in a real-time program window inference system using a similar rule to the standard rule pre seen earlier. For example, consider the following application of the pre opening rule.

$$\frac{\text{pre } level(\tau) > limit \Vdash \boxed{C} \sqsubseteq C'}{\star\{level > limit\}; \boxed{C} \sqsubseteq \star\{level > limit\}; C}$$

5 Supporting specifications

A real-time specification consists of a *context* stating any *assumptions* that the program can make about the variables in the environment and a specification command, written $\star\tilde{x} : [R]$, consisting of

- a vector of variables, \tilde{x} , called the *frame*, which may be modified by the command. This must be a subset of the *program variables* (outputs and local variables).
- a predicate, R , that states the *effect* that the command is to achieve. It may refer to variables and constants in the environment, and the initial and final values of the time variable (τ_0 and τ).

For example, the specification command for the program controlling the pump is written:

$$\star pump : [level(\tau) \leq limit \wedge (\forall t : [-\tau_0 \dots \tau] \bullet level(t) \geq minlevel)].$$

That is, modify pump so that the level stays above the minimum level for the active duration of the statement, but is below the limit when the statement finishes.

References to variables in the effect often refer to the start and finish times τ_0 and τ . The convention used above for assertions is extended so that unindexed references to time-dependent variables v and v_0 stand for $v(\tau)$ and $v(\tau_0)$ in the effect.

Definition 2 (at times) For a predicate or expression R , $R@(x, y)$ is defined to be R with all unindexed occurrences of the form v_0 replaced by $v(x)$ and all unindexed occurrences of v replaced by $v(y)$, and as well, all occurrences of τ_0 and τ in R are replaced by x and y respectively.

Note that the substituted predicate $P@(\tau_0, \tau)$ must not, for example, contain any free occurrences of τ_0 if P contains no occurrences of τ_0 or zero-subscripted variables. Hence $\tau_0 \text{ nfi } P@(\tau_0, \tau)$ may be true, where $x \text{ nfi } F$ stands for ‘ x does not occur free in F ’.

5.1 Semantics

The real-time specification command is defined in terms of the standard specification command using conventions in which only the time variable is updated [UF96]. The equivalent standard specification command allows time to increase and insists that all program variables that are not in the frame remain stable (unchanged) for the duration of the command.

Definition 3 (stable) Given a variable, v , and a set of times, S ,

$$stable(v, S) \hat{=} (\forall t, u : S \bullet v(t) = v(u)).$$

Stable is frequently overloaded by applying it to a vector of variables. In this case it should be taken to mean the conjunction of the stability condition applied to each of the variables in the vector, i.e.,

$$stable(\vec{x}, S) = stable(x_1, S) \wedge \dots \wedge stable(x_n, S).$$

The semantics of the standard refinement calculus is given in terms of predicate transformers.

$$PTrans \hat{=} Pred \rightarrow Pred$$

For a given environment, $\rho : Env$, the semantics of a real-time command describes a predicate transformer from *effects* to *assumptions* with respect to ρ .

$$\mathcal{M}_\rho : Command \rightarrow PTrans$$

The real-time assertion can now be defined more formally.

Definition 4 (assertion)

$$\mathcal{M}_\rho(\star\{P\}) \triangleq \{P@_\tau\}$$

where the right side stands for the predicate transformer corresponding to the standard refinement calculus command.

The specification command can now be defined [UF96]. The initial value τ_0 is explicitly introduced as a logical constant [Mor94, pp. 69]. We use $\hat{\rho}$ to denote the program variables in the environment ρ , i.e., $\hat{\rho} \triangleq \rho.Output \cup \rho.Var$.

Definition 5 (specification)

$$\mathcal{M}_\rho(\star\tilde{x} : [R]) \triangleq |[\mathbf{con} \tau_0 : Time \bullet \{\tau_0 = \tau\}; \\ \tau : [R@(\tau_0, \tau) \wedge \tau_0 \leq \tau \wedge stable(\hat{\rho}\backslash\tilde{x}, [-\tau_0 \dots \tau -])]]|$$

where $\hat{\rho}\backslash\tilde{x}$ stands for the program variables of the environment with all elements of the frame \tilde{x} removed.

5.2 Strengthen effect

The window opening rule post encodes the strengthen-effect rule from the refinement calculus. Producing such a rule for the real-time refinement calculus is complicated by the extra definitions in the meaning of the specification command. However, a similar rule is produced by combining several standard opening rules together. The final rule is presented first.

Opening Rule 6 (strengthen effect) For u a fresh name,

$$\begin{array}{l} \text{pre } P[\tau_0, \tau \backslash u, \tau_0] \wedge \tau_0 \leq \tau \wedge \\ \quad stable(\hat{\rho}\backslash\tilde{x}, [-\tau_0 \dots \tau -]) \\ \text{inv } I[\tau_0 \backslash u] \\ \text{lval } L[\tau_0 \backslash u] \\ \text{con } \tau_0 : Time \end{array} \quad \Vdash \boxed{R@(\tau_0, \tau)} \Leftarrow R'@(\tau_0, \tau)$$

$$\text{pre } P; \text{ inv } I; \text{ lval } L \Vdash \star\tilde{x} : [\boxed{R}] \sqsubseteq \star\tilde{x} : [R']$$

The strengthen effect law can be used to replace an effect R by another R' under the assumptions P with time increasing from τ_0 to τ and all variables not in the frame being stable. The fresh constant u hides old occurrences of τ_0 in the original hypotheses. The new τ_0 replaces τ in the precondition as τ refers to the post-state in the effect $R@(\tau_0, \tau)$. This rule is built from a combination of the standard program window inference rules as follows. Starting from the refinement

$$\text{pre } P; \text{ inv } I; \text{ lval } L \Vdash \star \tilde{x} : [R] \sqsubseteq \star \tilde{x} : [R']$$

and expanding the definition of the real-time specification command we get:

$$\begin{array}{l} \text{pre } P \quad |[\mathbf{con} \tau_0 : \mathit{Time} \bullet \{\tau_0 = \tau\}; \\ \quad \tau : [R@(\tau_0, \tau) \wedge \tau_0 \leq \tau \wedge \mathit{stable}(\hat{\rho} \setminus \tilde{x}, [-\tau_0 \dots \tau -])]]| \\ \text{inv } I \Vdash \sqsubseteq \\ \text{lval } L \quad |[\mathbf{con} \tau_0 : \mathit{Time} \bullet \{\tau_0 = \tau\}; \\ \quad \tau : [R'@(\tau_0, \tau) \wedge \tau_0 \leq \tau \wedge \mathit{stable}(\hat{\rho} \setminus \tilde{x}, [-\tau_0 \dots \tau -])]]| \end{array}$$

Opening on the constant introduction using rule \mathbf{con} (Figure 1) with a fresh u .

$$\begin{array}{l} \text{pre } P[\tau_0 \setminus u] \quad \{\tau_0 = \tau\}; \\ \text{inv } I[\tau_0 \setminus u] \quad \tau : [R@(\tau_0, \tau) \wedge \tau_0 \leq \tau \wedge \mathit{stable}(\hat{\rho} \setminus \tilde{x}, [-\tau_0 \dots \tau -])] \\ \text{lval } L[\tau_0 \setminus u] \quad \Vdash \sqsubseteq \\ \text{con } \tau_0 : \mathit{Time} \quad \{\tau_0 = \tau\}; \\ \quad \tau : [R'@(\tau_0, \tau) \wedge \tau_0 \leq \tau \wedge \mathit{stable}(\hat{\rho} \setminus \tilde{x}, [-\tau_0 \dots \tau -])] \end{array}$$

Applying rule \mathbf{pre} .

$$\begin{array}{l} \text{pre } P[\tau_0 \setminus u] \wedge \tau_0 = \tau \\ \text{inv } I[\tau_0 \setminus u] \\ \text{lval } L[\tau_0 \setminus u] \\ \text{con } \tau_0 : \mathit{Time} \\ \Vdash \\ \tau : [R@(\tau_0, \tau) \wedge \tau_0 \leq \tau \wedge \mathit{stable}(\hat{\rho} \setminus \tilde{x}, [-\tau_0 \dots \tau -])] \\ \sqsubseteq \\ \tau : [R'@(\tau_0, \tau) \wedge \tau_0 \leq \tau \wedge \mathit{stable}(\hat{\rho} \setminus \tilde{x}, [-\tau_0 \dots \tau -])] \end{array}$$

Applying rule `post` with a fresh τ' .

$$\begin{array}{l}
\text{pre}(P[\tau_0 \setminus u] \wedge \tau_0 = \tau)[\tau \setminus \tau'] \\
\text{inv } I[\tau_0 \setminus u] \\
\text{lval } L[\tau_0 \setminus u] \\
\text{con } \tau_0 : \textit{Time} \\
\vdash \\
R@(\tau_0, \tau) \wedge \tau_0 \leq \tau \wedge \textit{stable}(\hat{\rho} \setminus \tilde{x}, [-\tau_0 \dots \tau -]) \\
\Leftarrow \\
R'@(\tau_0, \tau) \wedge \tau_0 \leq \tau \wedge \textit{stable}(\hat{\rho} \setminus \tilde{x}, [-\tau_0 \dots \tau -])
\end{array}$$

The rule `post` renames occurrences of τ to the fresh name τ' within the `pre` context because within the postcondition τ refers to the post-state, not the pre-state. In this case we have (after substitution) that $\tau_0 = \tau'$, and hence the variable τ' is superfluous – we can use τ_0 everywhere instead, giving us $P[\tau_0, \tau \setminus u, \tau_0]$. The final rule 6 is obtained by focussing on $R@(\tau_0, \tau)$.

5.3 Refinement rules for simple commands

The meaning of some of the simple real-time refinement commands is given by the following definitions in which we write $C \hat{=}_{\rho} C'$ for $\mathcal{M}_{\rho}(C) \hat{=} \mathcal{M}_{\rho}(C')$.

Definition 7 (simple commands)

$$\begin{array}{l}
\text{idle} \hat{=}_{\rho} \star[\textit{true}] \\
\text{read}(v, x) \hat{=}_{\rho} \star x : [x \in v \text{ (} [-\tau_0 \dots \tau -] \text{)}] \\
\text{deadline } D \hat{=}_{\rho} \star[\tau_0 = \tau \wedge \tau \leq D] \\
\tilde{x} := \tilde{D} \hat{=}_{\rho} \star \tilde{x} : [(\tilde{x} @ \tau) = (\tilde{D} @ \tau_0)]
\end{array}$$

`idle` advances time but does nothing else. Remember, $\star[\textit{true}]$ implicitly includes $\tau_0 \leq \tau \wedge \textit{stable}(\hat{\rho}, [-\tau_0 \dots \tau -])$.

`read`(v, x) puts the value read from input variable v into program variable x . The value input must be observed during the read.

`deadline` D allows a time deadline to be set at the absolute time D . D can make no reference to τ_0 , i.e., $\tau_0 \textit{nfi } D@(\tau_0, \tau)$.

assignment is written $\tilde{x} := \tilde{D}$. The vector of expressions, \tilde{D} , must be time independent and $\tau_0 \textit{nfi } \tilde{D}@(\tau_0, \tau)$.

Other commands such as `skip`, `gettime`, and `delay until` are defined by Utting and Fidge [UF96]. A set of laws for introducing and transforming

these commands is given by Hayes and Utting [HU97]. A single example is given here to illustrate the use of program window inference. Rules for the other commands follow similarly.

The `read` command illustrates the use of `lval` constraints. The premiss of the refinement rule for the introduction of the `read` command requires that the variable v must be in the set *Input* and the variable x must be from the set *Output* or the set *Var* and of a type compatible with v , where $Type(v)$ gives the declared type of the variable v .

Refinement Rule 8 (read)

$$\frac{\begin{array}{l} \text{lval } x \in \text{Output} \cup \text{Var} \\ \text{lval } v \in \text{Input} \\ \text{inv } Type(v) \subseteq Type(x) \end{array}}{\star x : [x \in v \ll [-\tau_0 \dots \tau -] \gg] \sqsubseteq \text{read}(v, x)}$$

5.4 Sequential composition

Sequential composition is defined using the composition (\circ) of predicate transformers defined in the standard refinement calculus.

Definition 9 (sequential composition) *Given an environment ρ ,*

$$\mathcal{M}_\rho(C ; D) \triangleq (\mathcal{M}_\rho(C)) \circ (\mathcal{M}_\rho(D))$$

The refinement rule for introducing sequential composition implements the effect R in the specification by first achieving some other intermediate effect Q and then achieving the effect R . The logical constant u is used to equate the start time of the implementation with that of the specification.

Refinement Rule 10 (sequential composition)

$$\frac{\begin{array}{l} \text{lval } u \text{ nfi } R \\ \text{lval } u \text{ nfi } \tilde{x} \end{array}}{\begin{array}{l} \star \tilde{x} : [R] \\ \sqsubseteq \\ \ll [\mathbf{con } u : \text{Time} \bullet \{u = \tau\} ; \star \tilde{x} : [Q] ; \star \tilde{x} : [R@(\mathbf{u}, \tau)]] \gg \end{array}}$$

Example 1 We can implement the effect ‘delay for two time units’ by introducing a sequential composition of two single time unit delays.

$$\frac{\text{lval } u \text{ nfi } (\tau_0 + 2 \leq \tau)}{\begin{array}{l} \star[\tau_0 + 2 \leq \tau] \\ \sqsubseteq \\ \ll [\mathbf{con } u : \text{Time} \bullet \{u = \tau\} ; \star[\tau_0 + 1 \leq \tau] ; \star[u + 2 \leq \tau]] \gg \end{array}}$$

The second command refines to $\star[\tau_0 + 1 \leq \tau]$ by assuming the strongest post-condition of the first command (see Section 5.5).

Another version of the sequential composition rule may be used to introduce a deadline at time D . The time-valued expression D must contain no reference to τ_0 or zero-subscripted variables.

Refinement Rule 11 (separate deadline)

$$\frac{\begin{array}{l} \text{lval } \tau_0 \text{ nfi } D@(\tau_0, \tau) \\ \text{inv } D@ \tau \in \text{Time} \end{array}}{\star \tilde{x} : [R \wedge \tau \leq D] \sqsubseteq \star \tilde{x} : [R]; \text{deadline } D}$$

Example 2 We partially implement the specification of the pump controller with a program which uses a sensor placed within the sump to detect when the water level falls below a fixed limit. The limit must be placed high enough to allow *delay* seconds for the pump to stop before the water level reaches *minlevel*. We just consider the component that turns the pump off. The specification is defined by the following window. We have included only relevant information in the context.

$$\begin{array}{l} \text{input } \textit{sensor} : \textit{Boolean} \\ \text{output } \textit{pump} : \textit{Boolean} \\ \text{const } \textit{delay} : \textit{Time} \\ \text{con } \textit{chngs} : \textit{Time} \\ \text{pre } \textit{pump}(\tau) \wedge (\forall t : [-\tau \dots \textit{chngs} + \textit{delay} -]) \bullet \textit{sensor}(t) \Leftrightarrow \textit{chngs} \leq t \\ \Vdash \\ \star \textit{pump} : [\textit{pump}([- \tau_0 \dots \textit{chngs} -]) = \{ \textit{true} \} \wedge \neg \textit{pump} \wedge \\ \tau \leq \textit{chngs} + \textit{delay}] \sqsubseteq ? \end{array}$$

The precondition is that the pump is on and that *chngs* is the time that the *sensor* becomes *true*, i.e., *level* goes below limit. The effect we desire is that the pump is left on until *chngs* and then turned off within *delay*.

The first step is to separate out the timing constraint. We apply refinement rule 11 to separate the deadline in the above context. The following instance of the rule is found by unification.

$$\frac{\begin{array}{l} \text{lval } \tau_0 \text{ nfi } \textit{chngs} + \textit{delay} \\ \text{inv } \textit{chngs} + \textit{delay} \in \textit{Time} \end{array}}{\star \textit{pump} : [\textit{pump}([- \tau_0 \dots \textit{chngs} -]) = \{ \textit{true} \} \wedge \neg \textit{pump} \wedge \\ \tau \leq \textit{chngs} + \textit{delay}] \sqsubseteq ?}$$

$$\begin{array}{l} \sqsubseteq \\ \star \textit{pump} : [\textit{pump}([- \tau_0 \dots \textit{chngs} -]) = \{ \textit{true} \} \wedge \neg \textit{pump}]; \quad \text{(i)} \\ \text{deadline } \textit{chngs} + \textit{delay} \quad \text{(ii)} \end{array}$$

The proof obligations are discharged using information in the context.

5.5 Opening rules for sequential composition

The opening rules for sequential composition are the same as those for the standard calculus. Opening on the first command changes nothing in the context.

Opening Rule 12 (seq1)

$$\frac{\Gamma; \text{pre } P; \text{inv } I; \text{lval } L \Vdash \boxed{S} \sqsubseteq S'}{\Gamma; \text{pre } P; \text{inv } I; \text{lval } L \Vdash \boxed{S}; T \sqsubseteq S'; T}$$

However, when opening on the second command, we cannot assume the original precondition P . Instead, we assume the strongest postcondition of the first command and P as the precondition.

Opening Rule 13 (seq2)

$$\frac{\Gamma; \text{pre } sp(S, P); \text{inv } I; \text{lval } L \Vdash \boxed{T} \sqsubseteq T'}{\Gamma; \text{pre } P; \text{inv } I; \text{lval } L \Vdash S; \boxed{T} \sqsubseteq S; T'}$$

The strongest postconditions of some of the real-time commands are given below. We assume $\tau_0 \text{ nfi } P@(\tau_0, \tau)$, otherwise a fresh name must be used instead of τ_0 .

$$\begin{aligned} sp(\text{deadline } D, P) &\hat{=} P@_{\tau} \wedge \tau \leq D \\ sp(S; T, P) &\hat{=} sp(T, sp(S, P)) \\ sp(\text{idle}, P) &\hat{=} \exists \tau_0 \bullet P@_{\tau_0} \wedge \tau_0 \leq \tau \wedge \text{stable}(\hat{\rho}, [-\tau_0 \dots \tau -]) \\ sp(\star \tilde{x} : [R], P) &\hat{=} \exists \tau_0 \bullet P@_{\tau_0} \wedge \tau_0 \leq \tau \wedge \text{stable}(\hat{\rho} \setminus \tilde{x}, [-\tau_0 \dots \tau -]) \wedge \\ &\quad R@(\tau_0, \tau) \end{aligned}$$

Example 3 We focus on the second command in Example 1 using a combination of the opening rule for a logical constant block (con), opening rule for preconditions (pre), and opening rule 13 for sequential composition.

$$\begin{aligned} &\text{con } u : \text{Time} \\ &\text{pre } \exists \tau_0 \bullet (u = \tau)@_{\tau_0} \wedge \tau_0 \leq \tau \wedge \text{stable}(\hat{\rho}, [-\tau_0 \dots \tau -]) \wedge \\ &\quad (\tau_0 + 1 \leq \tau)@(\tau_0, \tau) \\ &\Vdash \\ &\star[u + 2 \leq \tau] \sqsubseteq ? \end{aligned}$$

The precondition can be simplified by applying Definition 2 and the one-point rule, and deleting useless predicates, leaving, $u + 1 \leq \tau \wedge \text{stable}(\hat{\rho}, [-u \dots \tau -])$. We can then simplify the specification using opening rule 6 for strengthen effect.

$$\frac{\begin{array}{l} \text{con } u, \tau_0 : \text{Time} \\ \text{pre } u + 1 \leq \tau_0 \wedge \text{stable}(\hat{\rho}, [-u \dots \tau_0 -]) \wedge \\ \quad \tau_0 \leq \tau \wedge \text{stable}(\hat{\rho}, [-\tau_0 \dots \tau -]) \\ \text{ft } u + 2 \leq \tau \Leftarrow \tau_0 + 1 \leq \tau \end{array}}{\star[u + 2 \leq \tau] \sqsubseteq \star[\tau_0 + 1 \leq \tau]}$$

The proof obligation is discharged by $u + 1 \leq \tau_0 \wedge \tau_0 + 1 \leq \tau \Rightarrow u + 2 \leq \tau$.

5.6 Assignment

The evaluation of the expression in an assignment command takes time. In order to make the assignment command deterministic, we insist that the time taken to evaluate the expression does not affect the value of the expression, i.e., the expression is time-independent.

If the value of an expression D is unaffected by changes to variables \tilde{x} over time, then it is said to be a *frame_stable* with respect to \tilde{x} . This is formalised by the following definitions [HU97].

Definition 14 (frame stable expression) *Given an environment, ρ , and a frame, \tilde{x} , such that $\tilde{x} \subseteq \hat{\rho}$, then an expression D , is frame stable with respect to \tilde{x} , if and only if*

$$\forall \tau_0, \tau : \text{Time} \bullet \tau_0 \leq \tau \wedge \text{stable}(\hat{\rho} \setminus \tilde{x}, [-\tau_0 \dots \tau -]) \Rightarrow (D @ \tau_0) = (D @ \tau)$$

As a shorthand we write *frame_stable_exp*(D, \tilde{x}).

An *idle_stable* expression remains stable over the execution of an idle command.

Definition 15 (idle stable expression) *Given an environment ρ , then an expression, D , is idle stable, written *idle_stable_exp*(D), if D is a frame stable expression with respect to the empty frame.*

The assignment statement rule is now stated.

Refinement Rule 16 (assignment) For u a fresh name,

$$\begin{array}{l}
\text{pre } P[\tau_0, \tau \setminus u, \tau_0] \wedge \tau_0 \leq \tau \wedge \\
\quad \text{stable}(\hat{\rho} \setminus \tilde{x}, [\neg \tau_0 \dots \tau \neg]) \\
\text{inv } I[\tau_0 \setminus u] \\
\text{lval } L[\tau_0 \setminus u] \\
\text{con } \tau_0 : \text{Time}
\end{array}
\quad \Vdash \quad
\begin{array}{l}
\text{lval } \tau_0 \text{ nfi } \tilde{D} @ (\tau_0, \tau) \\
\text{lval } \text{Con nfi } \tilde{D} \\
\text{pre } \text{idle_stable_exp}(\tilde{D}) \\
\text{pre } R @ (\tau_0, \tau) \Leftarrow \tilde{x} @ \tau = \tilde{D} @ \tau_0
\end{array}$$

$$\text{pre } P; \text{inv } I; \text{lval } L \Vdash \star \tilde{x} : [R] \sqsubseteq \tilde{x} := \tilde{D}$$

Here, and in the sequel, we have overloaded *nfi* by applying it to a set of identifiers, i.e., *Con nfi* \tilde{D} means ‘no name in the set *Con* occurs free in \tilde{D} ’.

A useful simplifying property is that an expression is *idle_stable* if it makes no reference to the time τ nor to any input variables.

Transformation rule 17 (idle stable property)

$$\begin{array}{l}
\text{lval } \tau \text{ nfi } \tilde{D} \\
\text{lval } \text{Input nfi } \tilde{D}
\end{array}
\quad \frac{}{} \quad
\text{pre } \text{idle_stable_exp}(\tilde{D})$$

Example 4 We refine the specification of the first component (i) of Example 2 to one which waits for the sensor to change and then turns off the pump. Opening a window on the first statement of the previous composition does not change the context. We then apply refinement rule 10 for sequential composition. After a process of simplification similar to Example 3 we get the following specification.

$$\star \text{pump} : [\text{pump}([\neg \tau_0 \dots \text{chngs} \neg]) = \{\text{true}\}] ; \star \text{pump} : [\neg \text{pump}]$$

We implement turning the pump off with an assignment statement.

$$\star \text{pump} : [\neg \text{pump}] \sqsubseteq \text{pump} := \text{false}$$

Applying refinement rule 16 and using rule 17, the idle stable property, gives the following proof obligations which can all be discharged without needing to refer to the context.

$$\begin{array}{l}
\text{lval } \tau_0 \text{ nfi } \text{false} \\
\text{lval } \text{Con nfi } \text{false} \\
\text{lval } \tau \text{ nfi } \text{false} \\
\text{lval } \text{Input nfi } \text{false} \\
\text{pre } \neg \text{pump}(\tau) \Leftarrow \text{pump}(\tau) = \text{false}
\end{array}$$

5.7 Variable introduction

Introducing a local variable may require time delays for the allocation and deallocation of the timed local variables.

Definition 18 (local block) *Given an environment, ρ , and a fresh variable v' not occurring in ρ , and provided T is nonempty, for all goal predicates G over the variables in ρ ,*

$$\mathcal{M}_\rho(|[\mathbf{var} \ v : T \bullet C]|)(G) \hat{=} (\forall v : \mathit{Time} \rightarrow T \bullet \mathcal{M}_{\rho'}(\mathit{idle}; C; \mathit{idle})(G[v \setminus v']))$$

where ρ' is the environment ρ with v renamed to v' and the set of local variables extended by v .

From this definition we get the following refinement rule, assuming the theory of types supports a definition of *nonempty*.

Refinement Rule 19 (introduce variable I)

$$\frac{\begin{array}{l} \mathit{lval} \ v \ \mathit{nfi} \ \tilde{x} \\ \mathit{lval} \ v \ \mathit{nfi} \ R \\ \mathit{inv} \ \mathit{nonempty}(T) \end{array}}{\mathit{idle}; \star \tilde{x} : [R]; \mathit{idle} \sqsubseteq |[\mathbf{var} \ v : T \bullet \star v, \tilde{x} : [R]]|}$$

However, this rule is only useful in the context where surrounding idle commands are available. To refine a command $\tilde{x} : [R]$ we need a way to introduce idle commands before and after it. We can derive a simple rule to prefix an idle if the effect is impervious to the specification command being prefixed by an idle command. A suitable condition on R is that it is *pre_idle_stable*.

Definition 20 (pre_idle_stable) *Given an environment ρ , a predicate R is pre-idle-stable, if and only if*

$$\forall u, \tau_0, \tau : \mathit{Time} \bullet u \leq \tau_0 \leq \tau \wedge \mathit{stable}(\hat{\rho}, [-u \dots \tau_0 -]) \wedge R@(\tau_0, \tau) \Rightarrow R@(u, \tau)$$

As a shorthand we write *pre_idle_stable*(R).

Refinement Rule 21 (idle before)

$$\frac{\mathit{pre} \ \mathit{pre_idle_stable}(R)}{\star \tilde{x} : [R] \sqsubseteq \mathit{idle}; \star \tilde{x} : [R]}$$

Similarly, a simple rule for adding an idle after a command can be derived if the effect is impervious to the command being postfixed by an idle.

Definition 22 (post_idle_stable) *Given an environment ρ , a predicate R is post-idle-stable, if and only if*

$$\forall u, \tau_0, \tau : \text{Time} \bullet \\ u \leq \tau_0 \leq \tau \wedge \text{stable}(\hat{\rho}, [\neg \tau_0 \dots \tau \neg]) \wedge R@ (u, \tau_0) \Rightarrow R@ (u, \tau)$$

As a shorthand we write $\text{post_idle_stable}(R)$.

Refinement Rule 23 (idle after)

$$\frac{\text{pre } \text{post_idle_stable}(R)}{\star \tilde{x} : [R] \sqsubseteq \star \tilde{x} : [R] ; \text{idle}}$$

An alternate definition for *introduce variable* is now derived.

Refinement Rule 24 (introduce variable II)

$$\frac{\begin{array}{l} \text{lval } v \text{ nfi } \tilde{x} \\ \text{lval } v \text{ nfi } R \\ \text{inv } \text{nonempty}(T) \\ \text{pre } \text{pre_idle_stable}(R) \\ \text{pre } \text{post_idle_stable}(R) \end{array}}{\star \tilde{x} : [R] \sqsubseteq \llbracket \text{var } v : T \bullet \star v, \tilde{x} : [R] \rrbracket}}$$

Example 5 Returning to the sequential composition in Example 4, we implement the component which leaves the pump on until time chngs . We introduce a local variable sens with which to read the sensor.

$$\frac{\begin{array}{l} \text{lval } \text{sens} \text{ nfi } \text{pump} \\ \text{lval } \text{sens} \text{ nfi } (\text{pump}(\neg \tau_0 \dots \text{chngs} \neg)) = \{\text{true}\} \\ \text{inv } \text{nonempty}(\text{Boolean}) \\ \text{pre } \text{pre_idle_stable}(\text{pump}(\neg \tau_0 \dots \text{chngs} \neg)) = \{\text{true}\} \\ \text{pre } \text{post_idle_stable}(\text{pump}(\neg \tau_0 \dots \text{chngs} \neg)) = \{\text{true}\} \end{array}}{\begin{array}{l} \star \text{pump} : [\text{pump}(\neg \tau_0 \dots \text{chngs} \neg)] = \{\text{true}\} \\ \sqsubseteq \\ \llbracket \text{var } \text{sens} : \text{Boolean} \bullet \\ \star \text{sens}, \text{pump} : [\text{pump}(\neg \tau_0 \dots \text{chngs} \neg)] = \{\text{true}\} \rrbracket \quad \text{(iii)} \end{array}}$$

Discharging the *pre_idle_stable* obligation requires showing

$$\begin{aligned} \forall u, \tau_0, \tau : \text{Time} \bullet \\ u \leq \tau_0 \leq \tau \wedge \text{pump}(\llbracket \neg \tau_0 \dots \text{chngs} \rrbracket) = \{true\} \wedge \\ \text{stable}(\text{pump}, \llbracket \neg u \dots \tau_0 \rrbracket) \Rightarrow \\ \text{pump}(\llbracket \neg u \dots \text{chngs} \rrbracket) = \{true\}. \end{aligned}$$

This may be discharged because we know that *pump* = *true* in subinterval τ_0 to *chngs* inclusive, and *pump* is unchanging in the adjoining subinterval u to τ_0 inclusive, so *pump* must equal *true* in interval u to *chngs*. As τ_0 is contained in both subintervals the concatenation of the subintervals is also an interval on which *pump* = *true*.

The *post_idle_stable* obligation is discharged more easily as the effect is required only up to *chngs* and so idling longer will not negate that. Note that the original deadline (ii) limits the amount of idling allowable.

5.8 Opening rule for a local variable block

The opening rule for a local variable block adds the local variable to the environment and updates the context by the time taken for the variable allocation.

Opening Rule 25 (var) For a fresh w' ,

$$\frac{\begin{array}{l} \text{pre } sp(\text{idle}, P[w \setminus w']) \\ \text{inv } I[w \setminus w'] \\ \text{lval } L[w \setminus w'] \\ \text{var } w : T \end{array} \quad \Vdash \boxed{S} \sqsubseteq S'}{\begin{array}{l} \text{pre } P \\ \text{inv } I \quad \Vdash \llbracket \mathbf{var } w : T \bullet \boxed{S} \rrbracket \sqsubseteq \llbracket \mathbf{var } w : T \bullet S' \rrbracket \\ \text{lval } L \end{array}}$$

When opening on the body of a local variable block, the pre-state of the body does not correspond directly to the pre-state of the block because allocating a local variable may take time. Hence the pre-state of the body corresponds to the strongest postcondition of *idle* with respect to the pre-state of the block.

Example 6 Using opening rule 25 on the previous specification (iii).

```

...
var sens : Boolean
pre  $\exists \tau_0 : Time \bullet$ 
    pump( $\tau_0$ )  $\wedge \tau_0 \leq \tau \wedge stable(\{sens, pump\}, [\tau_0 \dots \tau]) \wedge$ 
    ( $\forall t : [\tau_0 \dots chngs + delay] \bullet sensor(t) \Leftrightarrow chngs \leq t$ )
 $\Vdash$ 
 $\star sens, pump : [pump([\tau_0 \dots chngs]) = \{true\}] \sqsubseteq ?$ 

```

This may be refined by a loop testing the sensor in a manner similar to that used in [HU98].

6 Conclusion

Window inference is now an accepted technique for supporting both program refinement and theorem proving within a single framework. Program window inference extends window inference by representing information about the program context via explicit l-value, precondition and invariant contexts. In this paper we have studied the use of program window inference to support the sequential real-time refinement calculus. Fortunately, this has ended up being reasonably straightforward, in theory at least.

For l-value information we have had to extend the l-value context with support for differentiating between input, output and local variables, as well as constants and logical constants. This can be done in the same way that standard program window inference differentiates variables from other identifiers. In addition, in the real-time calculus variables are traces over (or functions of) time. These can be represented directly via the invariant context.

Program window inference does not provide support for Morgan's initial value (zero-subscripted) variables. The intention is that the user of program window inference would explicitly introduce logical constants when initial values are needed. Unfortunately, for the real-time calculus *every* command needs to make reference to the initial time, τ_0 , even if it is only to state that time progresses: $\tau_0 \leq \tau$. This problem is overcome by defining the real-time specification command as a composition of standard constructs. The opening rule for a real-time specification can then be derived via the composition of standard opening rules.

The real-time calculus also introduces subtleties because the value of an expression/predicate can change without the program itself explicitly changing the values of any variables. This is because the value of inputs can vary over time independently of any action of the program, and because the value of τ changes with the progression of time. Hence we need

to be able to state properties about the stability of expressions/predicates over time. For example, we need to be able to state that the expression in an assignment statement is time independent. The statement of such properties using program window inference has been straightforward.

There are constructs in the real-time language, such as the allocation and deallocation of a local variable, that take time but otherwise do not affect the values of the program variables. To handle the passing of time in such cases, the real-time calculus introduces concepts of pre-idle-stable and post-idle-stable. Again these can be directly modelled using program window inference.

Overall the modelling of the sequential real-time refinement calculus using program window inference has been straightforward, although the complexities of the real-time calculus over the standard calculus do show through. As a side effect, the rigour required for expressing the constructs using program window inference has meant that we have in some cases ended up with more general rules than the equivalent rules earlier devised by Hayes and Utting [HU97].

7 Acknowledgements

The authors would like to thank David Carrington, Colin Fidge, and Axel Wabenhurst for their help in preparing this paper. This research is funded by ARC Large Grant A49702415, *Efficient development of verified concurrent real-time programs through tool support*.

References

- [Bac80] R.-J. Back. Correctness preserving program refinements: Proof theory and applications. *Tract 131, Mathematisch Centrum, Amsterdam*, 1980.
- [CHN⁺96] David Carrington, Ian Hayes, Ray Nickson, Geoffrey Watson, and Jim Welsh. A tool for developing correct programs by refinement. In He Jifeng, editor, *Proc. BCS 7th Refinement Workshop, Bath, UK*, Electronic Workshops in Computing, pages 1–17. Springer, 1996.
- [Gru96] J. Grundy. Transformational hierarchical reasoning. *The Computer Journal*, 39(4):291–302, 1996.
- [HU97] I. Hayes and M. Utting. A sequential real-time refinement calculus. Technical Report 97–33, Software Verification Research Centre, The University of Queensland, August 1997.
- [HU98] Ian Hayes and Mark Utting. Deadlines are termination. In David Gries and Willem-Paul de Roever, editors, *Programming Concepts and Methods (PROCOMET '98)*, pages 186–204. Chapman and Hall, 1998.
- [Mah92] B. Mahony. *The Specification and Refinement of Timed Processes*. PhD thesis, Department of Computer Science, The University of Queensland, 1992.

- [Mor94] C. Morgan. *Programming from Specifications*. Prentice Hall, 2nd edition, 1994.
- [MV90] Carroll Morgan and Trevor Vickers. Types and invariants in the refinement calculus. *Science of Computer Programming*, 14:281–304, 1990.
- [NH95] R. Nickson and I. Hayes. Program window inference. Technical report 95–29, Software Verification Research Centre, The University of Queensland, July 1995.
- [NH97] R. Nickson and I. Hayes. Supporting contexts in program refinement. *Science of Computer Programming 29*, pages 279–302, 1997.
- [RS93] P.J. Robinson and J. Staples. Formalising the hierarchical structure of practical mathematical reasoning. *Journal of Logic and Computation*, 3(1):47–61, 1993.
- [UF96] M. Utting and C. J. Fidge. A real-time refinement calculus that changes only time. In He Jifeng, John Cooke, and Peter Wallis, editors, *BCS-FACS Seventh Refinement Workshop*, Electronic Workshops in Computing. Springer-Verlag, 1996. <http://www.springer.co.uk/ewic/workshops/7RW/>.